

Interactive L-System Plant Visualization Using Unity GL Rendering

Xinjie Ye

Mathematics for Games And V/AR

MA VIRTUAL AND AUGMENTED REALITY

25/11/2025



Abstract

The L-system provides a compact and powerful formal method for modeling plant structures through recursive rewriting rules (Lindenmayer, 1968). This project implements an interactive L-system visualization tool using Unity, C#, and a fully custom GL-based vector rendering pipeline. To enhance interactivity, the system provides real-time parameter adjustment functionality through keyboard control and runtime user interface panels. Users can modify the iteration depth, angle offset, and a random angle variation parameter to generate more natural and biologically plausible plant forms. This real-time visualization and interactive tool enable users to understand how changes in the number of iterations, angles, and randomness affect the complexity and form of the generated structures.

Keywords: Lindenmayer, L-systems, Unity, C#, GL-based vector rendering

Features

The main feature of this system is that I have provided users with a UI interface and keyboard operation buttons that can be interactively controlled in real time during runtime. Users can simply modify the parameters displayed in the UI interface to influence the generated L-system, including eight plant presets, the number of iterations, angle deviation, line segment length, and random angle variation. These parameter panels can help users understand the working principle of the model intuitively. All plant definitions are stored in the JSON configuration file in the StreamingAssets directory of Unity. It is very convenient to modify and expand them without changing the underlying code.

I have set up a group of keyboard hotkeys to facilitate users' quick adjustment of experimental parameters. Q – Cycle through the eight plant presets; Up / Down Arrow – Increase or decrease the iteration depth; Left / Right Arrow – Apply a positive or negative angular offset; PageUp / PageDown – Adjust the magnitude of stochastic angle variation (Figure 1 & 2). These control options enable users to meaningfully alter the complexity, orientation and natural variations of plant structures during the execution process (Prusinkiewicz, 1986).

```
// Q Switch Plant
if (Input.GetKeyDown(KeyCode.Q))
{
    int next = CurrentPlant + 1;
    if (next > 8) next = 1;
    LoadAndDraw(next);
}

// ↑ ↓ Iteration count
if (Input.GetKeyDown(KeyCode.UpArrow))
{
    iterations++;
    Redraw();
}
if (Input.GetKeyDown(KeyCode.DownArrow))
{
    iterations = Mathf.Max(1, iterations - 1);
    Redraw();
}
```

Figure 2

```
// ← → Angle offset
if (Input.GetKeyDown(KeyCode.LeftArrow))
{
    angleOffset -= 2f;
    Redraw();
}
if (Input.GetKeyDown(KeyCode.RightArrow))
{
    angleOffset += 2f;
    Redraw();
}

// PageUp / PageDown = RandomAngleRange
if (Input.GetKeyDown(KeyCode.PageUp))
{
    randomAngleRange += 1f;
    Redraw();
}
if (Input.GetKeyDown(KeyCode.PageDown))
{
    randomAngleRange = Mathf.Max(0f, randomAngleRange - 1f);
    Redraw();
}
```

Figure 1

During the project's operation, the user interface panel provides a runtime interface to complement keyboard control. This panel includes the following editable fields (Figure

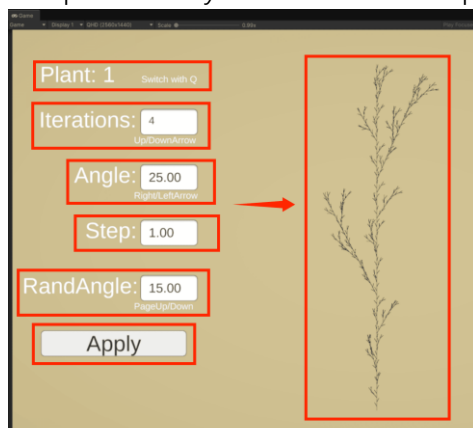


Figure 3

3): Current plant index, Iteration count, Base angle (including user offset), Random angle range, Step scaling factor. Click the "Apply" button to update the system immediately. The user interface and keyboard input are fully synchronized: whenever a plant is changed or a parameter is modified, all fields will automatically refresh to reflect the current status.

My model employs a JSON-based configuration system. Each plant is defined in a dedicated JSON file (Figure 4), which specifies: Axiom - the initial symbol, Angle - the base branching angle, Step - the line length unit, and Rules - a set of production rules mapping symbols to replacement strings. This structure matches standard deterministic context-free L-systems and fully supports variables, constants, and nested branching (Prusinkiewicz & Lindenmayer, 1990).

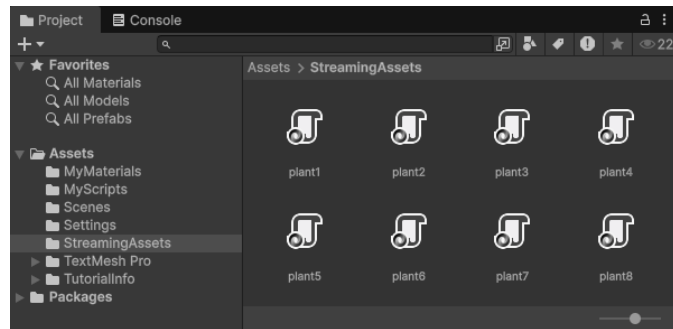


Figure 4

For the rendering of line segments, I utilized GL Vector Rendering. I didn't use the default LineRenderer of Unity, but instead employed a vector renderer based on custom GL. This method enables my model to generate clear and anti-aliased vector lines, avoiding the pixelation that occurs when magnified (Unity Technologies, 2023). Branching behavior is implemented through a stack-based turtle graphics interpreter, where each "[" saves the current state of the turtle and each "]" restores it (Smith, 1984).

In the camera view, I set up camera auto-adaptation. A camera auto-adaptation module continuously calculates the bounding box of the generated structure and adjusts the size of the orthographic camera (Figure 5 & 6). I use this setting to ensure that the plant is always centered and fully visible. This feature guarantees that the composition of the image remains consistent regardless of the iteration depth, angle offset, or random variations.

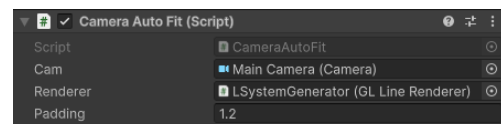


Figure 5

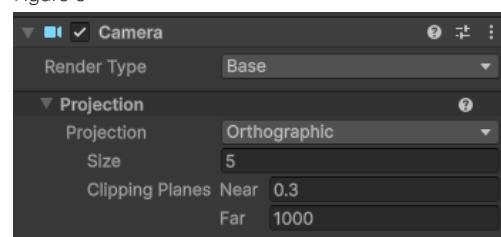


Figure 6

These features collectively provide a flexible and intuitive environment for exploring the procedural plant structures generated through L-system rules.

Data Classes

My model system mainly relies on several data structures to represent L-system

configurations, production rules, turtle states, and runtime parameters. It is the interaction among these different types of data that jointly enables the operation of my model.

The L-System configuration, in my project, each plant is defined in a JSON file, which is then loaded into the LSystemConfig class. This class contains (Figure 7): string axiom – the initial symbol of the system, float angle – the base branching angle, float step – the unit length used for line segments, 'Rule []' rules – the set of production rules. This structure reflects the standard specification of a deterministic context-free L-system (Prusinkiewicz & Lindenmayer, 1990).

```
[System.Serializable]
2 个引用
public class LSystemConfig
{
    public string axiom;
    public float angle;
    public float step;
    public Rule[] rules;
}

[System.Serializable]
2 个引用
public class Rule
{
    public string key;
    public string value;
}
```

Figure 7

In Production Rules, I respectively use string key to represent a single variable (e.g., "F" or "X"), and string value to control the replacement string applied during rewriting. My system supports: variables, constants, and nested branch symbols []. These match the original formalism of parallel rewriting systems (Lindenmayer, 1968).

The TurtleState structure, in the plant rendering process of the model, the system uses turtle graphics to interpret the generated string. I mainly use the TurtleState structure to store Vector3 position and Quaternion rotation. When encountering [the turtle's state is pushed to a stack; when encountering] the saved state is popped and restored

```
else if (c == '[')
{
    stack.Push(new TurtleState(pos, rot));
}
else if (c == ']')
{
    var st = stack.Pop();
    pos = st.pos;
    rot = st.rot;
}
```

Figure 8

(Figure 8). This stack-based interpretation was popularized for plant visualization in early graphics research (Smith, 1984).

The Generated Instruction String. The L-system output I am currently generating is stored as a standard C# string, which includes the following symbols (Figure 9): F – draw forward, + / - – rotate, [/] – push/pop turtle state. This instruction sequence is provided as input to the GL rendering stage.

```
foreach (char c in instructions)
{
    if (c == 'F')
    {
        Vector3 newPos = pos + rot * Vector3.right * step;
        glRenderer.AddLine(pos, newPos);
        pos = newPos;
    }
    else if (c == '+')
    {
        float rand = Random.Range(-angleRandomRange, angleRandomRange);
        rot *= Quaternion.Euler(0, 0, angle + rand);
    }
    else if (c == '-')
    {
        float rand = Random.Range(-angleRandomRange, angleRandomRange);
        rot *= Quaternion.Euler(0, 0, -(angle + rand));
    }
}
```

Figure 9

The Runtime Parameter Data. I put

the additional runtime parameters, such as iteration depth, angle offset, random angle range, step scale, and selected plant index, into the `LSystemController` for management. These values can be modified either by keyboard input or through the UI panel and are synchronized in every frame.

Main Structures

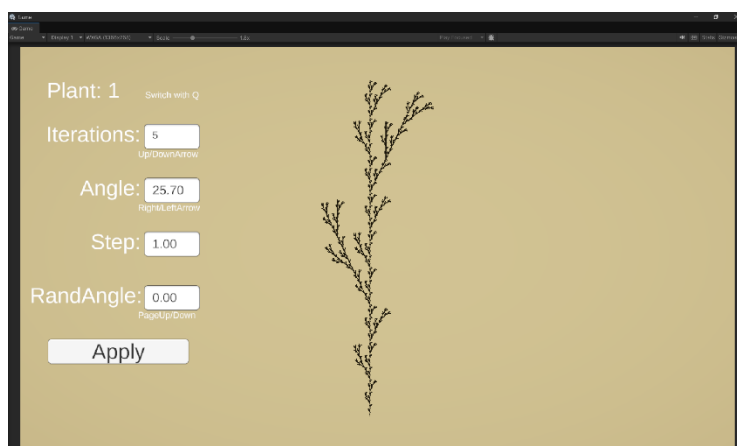
The overall workflow of this system follows the classic L-system processing flow - loading, rewriting, interpreting and rendering. This architecture is modular and consists of four core components: configuration loader, L-system generator, renderer, and controller.

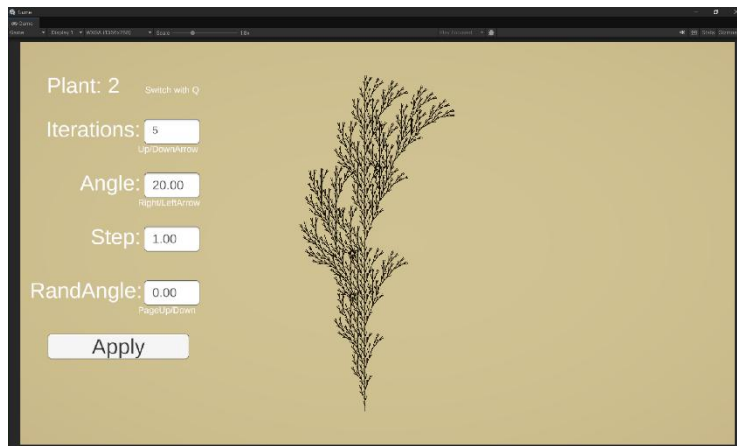
1. **Configuration Loading.** When the user switches plant presets by pressing the Q key, the system will load the corresponding JSON file from the `StreamingAssets` directory. Each file will be parsed into an `LSystemConfig` object, which contains the axiom, rewriting rules, angle, and step size. This design achieves seamless scalability without the need to modify the code. Even ordinary users can add new plants through simple parameter adjustments.
2. **String Rewriting Process.** The rewriting phase will be implemented in the `LSystemGenerator` class. Starting from the axiom, the generator applies all production rules in parallel for a user-selected number of iterations. The output of this process is a single instruction string composed of F, +, -, [, and] symbols, which encodes the final structural description of the plant.
3. **Turtle Interpretation.** The generated instruction string is interpreted using a 2D turtle graphics model, a method widely used in L-system visualization (Smith, 1984). The key behaviors include the rules of the symbols F, +, -, [, and] that have been mentioned before. This stack-based branching mechanism can create highly complex hierarchical structures. In addition, I have introduced random angle variations into my system. Each rotation operation may be randomly disturbed within the range controlled by the user. This concept is derived from the random plant modeling technique (Prusinkiewicz, 1986).
4. **GL-Based Rendering Pipeline.** I chose to use a custom GL vector renderer for rendering instead of Unity's `LineRenderer` component. Each interpreted line segment is submitted to the GL pipeline to generate clear, resolution-independent vector lines. This method avoids jagged artifacts and maintains visual clarity when the camera zooms, and is supported by the Unity GL API documentation (Unity Technologies, 2023).

5. Camera Auto-Fitting. After the rendering is completed, the system will calculate the bounding box of the generated plants and automatically adjust the size and position of the orthographic camera. This ensures that no matter how deep the iteration is or what random changes exist, the entire structure remains centered and fully visible at all times. Each time the user redraws the model, this camera calculation process runs once, providing the user with a smooth exploration experience.
6. Runtime Control Loop. The LSystemController script is responsible for coordinating all system activities, including detecting user input (keyboard or user interface), updating runtime parameters (iteration count, angle offset, random angle range, step ratio), triggering L-system regeneration and GL re-rendering, performing camera auto-adaptation, and refreshing the user interface to display the updated status. This loop is executed once per frame to ensure rapid and intuitive real-time interaction with the model.

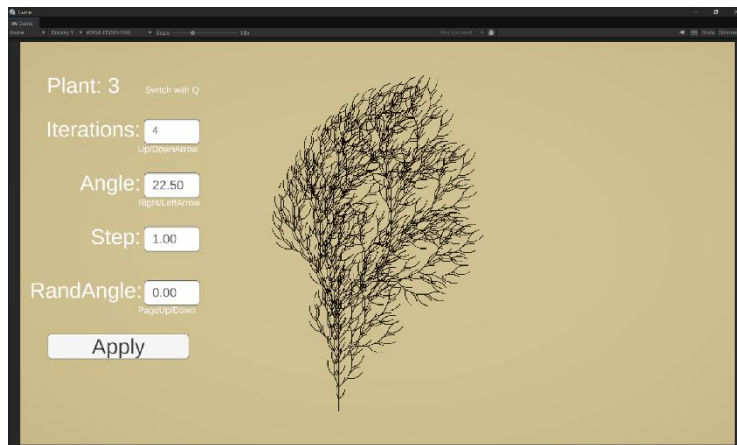
Results

The system I designed replicated all six classic examples specified in the course assignment and added two custom designs on this basis. Each model is defined by a unique set of generation rules, angles, and iteration depths. The structure generated by this system matches the expected form typically seen in classic L-system demonstrations in textbooks (Prusinkiewicz & Lindenmayer, 1990). The results are as follows (my program's rendering result is on the left, and the sample case is on the right):

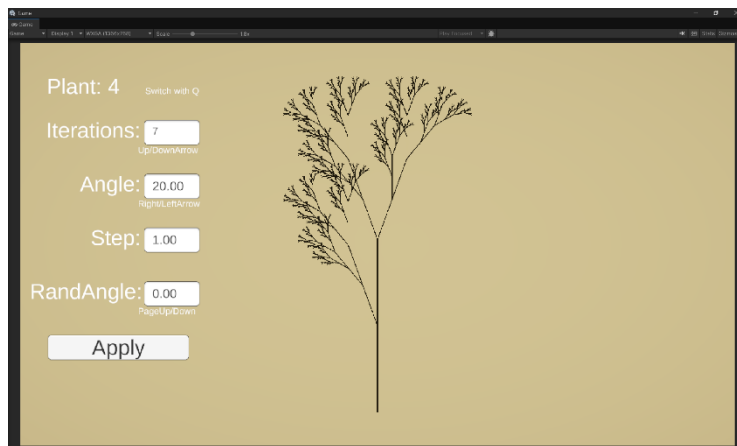




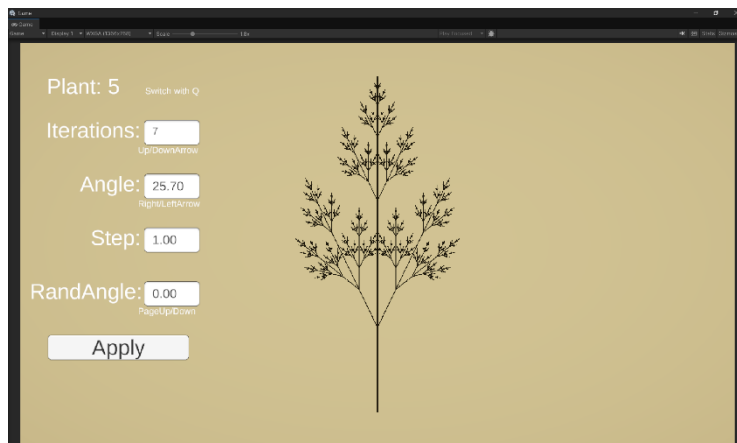
b
 $n=5, \delta=20^\circ$
 F
 $F \rightarrow F[+F]F[-F][F]$



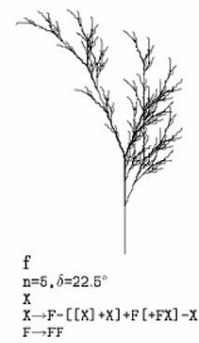
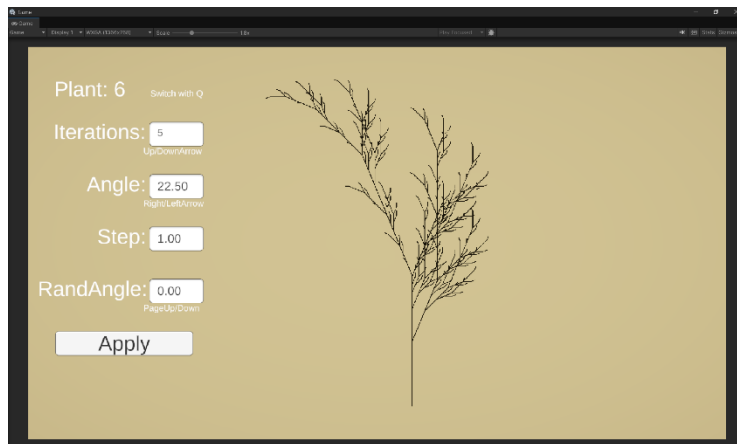
c
 $n=4, \delta=22.5^\circ$
 F
 $F \rightarrow FF[-F+F+F][+F-F]$



d
 $n=7, \delta=20^\circ$
 X
 $X \rightarrow F[+X]F[-X]+X$
 $F \rightarrow FF$



e
 $n=7, \delta=25.7^\circ$
 X
 $X \rightarrow F[+X][-X]FX$
 $F \rightarrow FF$



To extend the system beyond the examples required for the course, I created two additional plant definitions.

The branch angle of plant No. 7 is wider, forming an open fan-shaped structure (Figure 10).

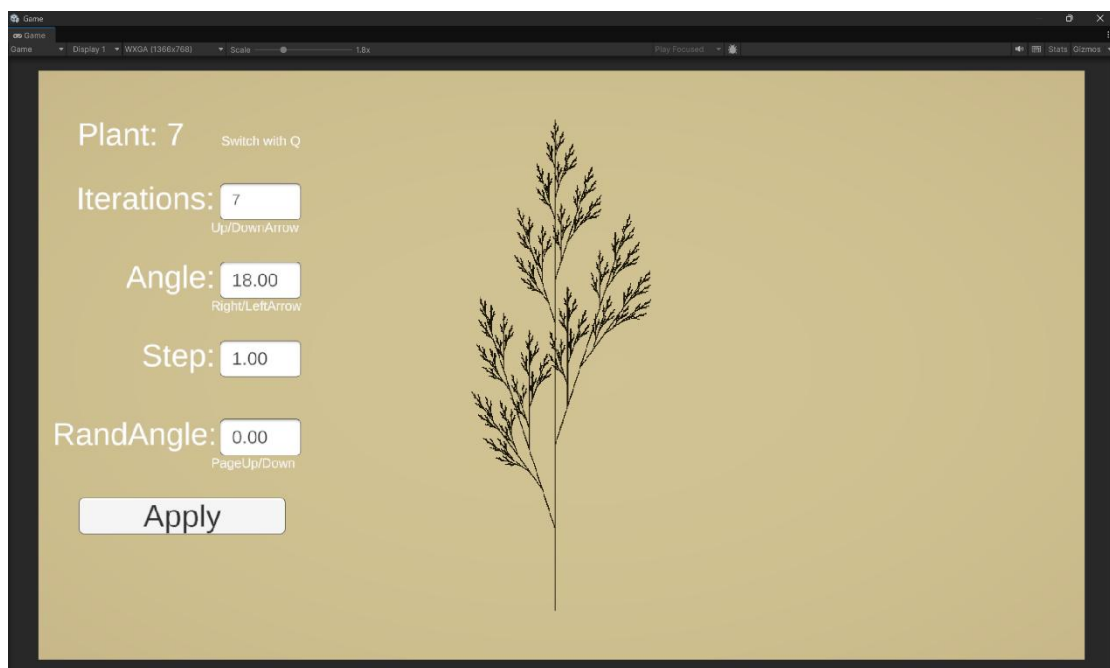


Figure 10

Plant No. 8 frequently branched at a relatively shallow depth, presenting a compact and shrub-like appearance (Figure 11).

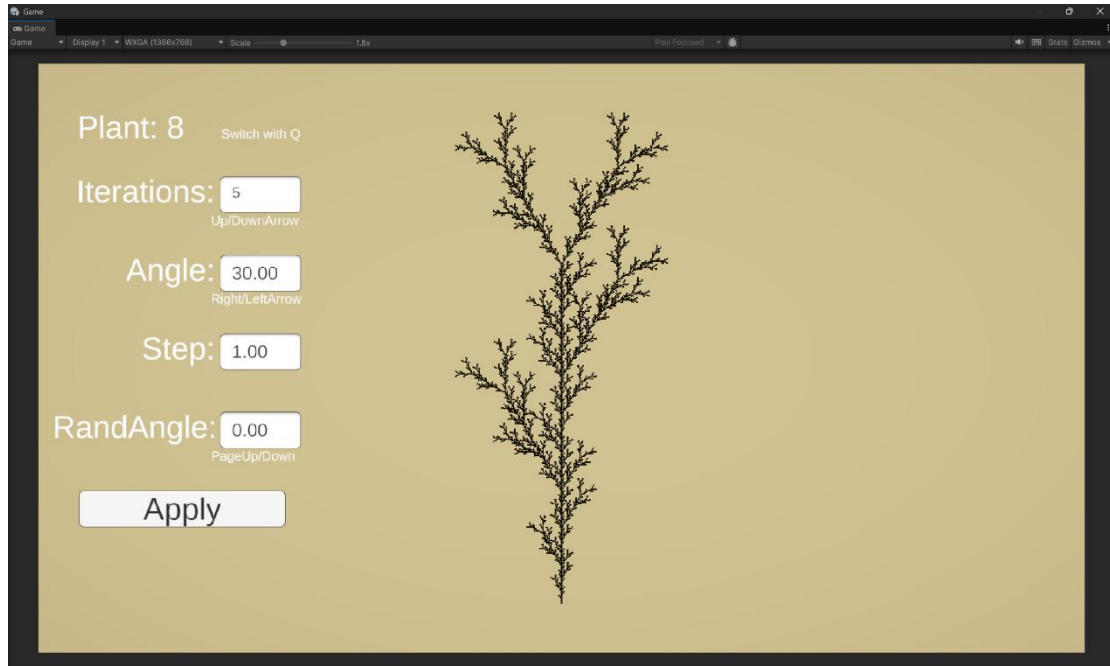


Figure 11

These two custom plants demonstrate the flexibility of the JSON configuration system and the renderer's ability to visualize various structures without any code modifications.

Conclusion

In conclusion, the project has successfully implemented an interactive L-system visualization tool using Unity, C#, and a fully customized GL vector rendering pipeline. I use this system to load plant definitions from JSON files and apply deterministic context-free rewriting rules. Then, use a turtle-based graphics model to explain the generated instruction string. All six required plant examples were accurately recreated, and two additional presets were also developed to demonstrate its scalability.

In my project, I also incorporated keyboard shortcuts and dedicated user interface panels for adding runtime parameter controls, allowing for a more intuitive exploration of L-system behavior through these parameter panels. Users can immediately observe how changes in iteration depth, branch angle, and random angle affect the geometry and complexity of the generated structure. The camera auto-adaptation module I developed can also ensure stable framing in all configurations. The GL renderer I use, on the other hand, offers clear and resolution-independent visual effects.

Overall, my system has achieved the goals I wanted to reach at present and has expanded the implementation of the standard L-system through interactive functions

and random variations. It enables users to visually present the behavioral patterns of L-System through simple adjustments of the parameters of the user interface.

References

Algorithmic Botany Group. (2024). L-system resources. <https://algorithmicbotany.org/>

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18(3), 280–299.

Prusinkiewicz, P. (1986). Graphical applications of L-systems. *Proceedings of Graphics Interface*, 247–253.

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. Springer.

Smith, A. R. (1984). Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics*, 18(3), 1–10.

Št'ava, O., Beneš, B., Mech, R., & Miller, G. (2010). Inverse Procedural Modeling of Trees. *Computer Graphics Forum*, 29(2), 529–538.

Ulam, S. (1962). On some mathematical properties of cell growth. *Journal of Theoretical Biology*, 3(1), 113–120.

Unity Technologies. (2023). Unity Manual: GL Class Reference. <https://docs.unity3d.com/>